XI. Focus

Formal Development of a Production Cell in Focus
A Case Study

Max Fuchs, Jan Philipps

Technische Universität München

Abstract

A specification for a production cell is developed using the design method Focus. The specification comprises both the components of the production cell and the corresponding control programs. This work investigates the suitability of a rule system for the stepwise refinement of distributed systems in an assumption/commitment style. As a running example we consider the elevating rotary table of a production cell. Besides descriptive and constructive specifications for this component an executable simulation program in a functional language is presented. All design steps except the final transformation to an executable program are proven correct.

11.1 Introduction

In the development of distributed systems, methodical issues and the formalism used are of great importance. This is because distributed systems are often very complex and there are strict requirements regarding correctness and reliability. The formalism used to specify the production cell is Focus [1]. The specification comprises both the components and the control programs of the cell.

As our main interest in specifying the production cell was the methodical aspect of stepwise refinement, we did not prove all the safety requirements of the task description. We did show, however, that all refinement steps are correct, and that the cell fulfills a basic liveness property: every part that enters the factory leaves it again.

The rest of this chapter is structured as follows:

- In section 12.2 we give an overview about Focus and introduce the formalism used for our specifications.
- In section 12.3 we develop specifications and a (functional) control program for the elevating rotary table of the production cell.
- In section 12.4 we discuss some shortcomings and extension of our approach, and describe how general safety properties could have been included and verified.

11.2 Method

Focus is a design method for distributed reactive systems. Focus provides models, formalism and verification principles for stepwise specification and development. The development process is split into three different abstraction levels. These levels are *requirement specification*, *design specification*, and *implementation*. Since all development steps take place on a formal basis, the overall correctness can be proven, if required.

The most abstract specification, which is called requirement specification, is formulated either using trace logic or stream processing functions. A trace specification gives an overview over the global system behaviour by describing the allowed sequences of actions in the system. A specification on the design level requires an interface description — for the production cell we do have an interface description and therefore we start at the design level.

At the level of design specifications, the system is modeled by a network of components that work concurrently and communicate over unbounded FIFO channels. Focus follows the tradition of Kahn [4] by describing the components as functions that map sequences of input messages to sequences of output messages (stream processing functions). System components have a clearly defined interface, and they can be developed in a modular way. Design decisions can be checked at the point they are taken, components can be refined independently, and already developed components can be reused in other systems.

It is at this level, where most of the developement in the Focus framework takes place. For this reason, Focus was recently extended with a rule system [7] similar to Hoare‘s well-known rules for sequential systems. The rules allow refinement of specifications formulated in the assumption/commitment-style.

The implementation level finally gives executable programs for the system‘s components. There are a number of possible target languages for Focus specification, ranging from abstract, not yet implemented languages like AL and PL [1], over other specification languages like SDL [8] to clocked hardware [10]. For the production cell we used Concurrent ML, a version of Standard ML enriched with threads and communication primitives.

11.2.1 Basic concepts

In this section we describe the formalism used at the design level of Focus.

For function application we write *f.x* instead of f*(x)* to save brackets. The dot is left-associative, i.e. *f.g.x = (f.g).x*.

The most important data structure in Focus is the stream. Streams are used to describe the communication history of the channels between the various components of a distributed system. A stream is a finite or infinite sequence of data elements called *messages*. Given a data set *D* we write:

- D^* for the set of finite streams over *D*,
- D^∞ for the set of infinite streams over *D* and
- D^ω for the set of all streams over *D* ($D^\omega = D^* \cup D^\infty$).

For the empty stream we write $\langle\rangle$ and for a finite stream of *n* elements over D we write $\langle d_1, d_2, ..., d_n\rangle$.

We define the following operators on streams. For $s, t \in D^\omega$ and $A \subseteq D$:

- *ft.s* denotes the first element of *s*, if *s* is non-empty.
- *rt.s* denotes *s* without its first element.
- *#s* denotes the number of elements in the stream *s*, or ∞ if *s* is infinite.
- *A©s* denotes the stream *s* with all elements not in *A* removed. For singleton sets $A = \{ a \}$ we write *a©s*.
- $s \circ t$ denotes the concatenation of *s* and *t*.
- s^n denotes the result of concatenating *s* with itself *n* times ($s^0 = \langle\rangle$).

We also define a partial order (“prefix order”) .∠. on streams via

$$s \angle t \equiv \exists\, u \in D^\omega : s \circ u = t$$

The set of all streams over a given data set D together with the prefix order forms a complete partial order with the empty stream as the least element, and the elements of D^∞ to guarantee the existence of least upper bounds.

We model components of distributed systems as stream-processing functions (SPF). A SPF is a function from tuples of input streams to tuples of output streams, that is monotonic and continuous w.r.t. the prefix order $\angle$. Monotonicity implies that a component cannot undo any output that has already been emitted. Because of continuity, a component's behaviour is fully described by its behaviour for finite inputs. Therefore components can work in parallel.

A simple SPF is the function *map*, which applies a given function f to every element of its input stream:

$$map.f.(\langle d_1 \rangle \circ s) = \langle f.d_1 \rangle \circ map.f.s$$

For composing SPF there are the three classical operators, namely sequential composition $(f ; g).x = g(f(x))$, parallel composition $(f \parallel g).(x,y) = (f.x, g.y)$, and a feedback operation μ.

Each composition yields again a SPF.

11.2.2 Specifications

We specify components by formulating relations between input and output of their SPFs. Generally, we don't specify the behaviour for all possible input streams, but only for those that fulfil certain expectations of the component. This style of specification is usually called assumption/commitment-style.

We write specifications as a pair $[A,C]$ where A (the assumption) is a predicate with the input streams as free variables, and C (the commitment) is a predicate with the input and output streams of the agent as free variables. The denotation of such a specification is the set

$$\{ f \in I^\omega \to O^\omega \mid \forall i \in I^\omega : A(i) \Rightarrow C(i,f.i) \}$$

of all functions f such that when the input i of f fulfils the assumption of the component, the output o of f is related to i according to the component's commitment. In this case, we considered a component that consumes input data of a set I and produces output of a set O, but it is easy to generalize this definition to arbitrary sets and tuples of input streams.

For example, we can specify a component that applies a function f to its input messages, where no particular properties of the input are required, as

$$[\ \textbf{true},\ o = map.f.i]$$

A specification $[A_2,C_2]$ is called a *refinement* of a specification $[A_1,C_1]$, if the denotation of $[A_2,C_2]$ is a subset of the denotation of $[A_1,C_1]$. We then write

$$[A_1,C_1] \succcurlyeq [A_2,C_2]$$

For each composition form of SPFs there is a corresponding decomposition rule in Focus. As an example, we introduce the composition form of a master/slave-system (Figure 1). Master/slave-systems can be used to describe the interaction of two components, where one controls the other. For instance, a control program sends commands to the controlled machine, which returns acknowledgment messages. Given two functions, f and g, the output of a master/slave-system $o = (f \oplus g).i$ for a given input i is defined by the least fixpoint of the following set of equations:

$$(o,y) = f(i,x)$$

$$z = g.y$$

$$x = z$$

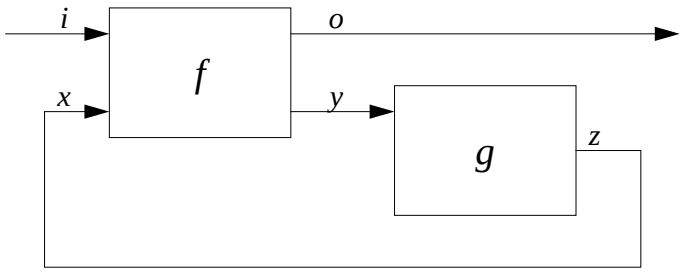


Figure 1 Master/Slave-System

Since the set of all streams forms a complete partial order, and the functions f and g are monotonic, least fixpoints always exist (they may be infinite). Moreover, because of the continuity of the functions, the least fixpoint can be computed by the stepwise communication between components.

The corresponding decomposition rule is shown in Figure 2. It allows us to refine a specification $[A,C]$ of the function $f \oplus g$ to specifications $[A_1,C_1]$ for the master and $[A_2,C_2]$ for the slave function. The rule is closely related to the WHILE-

rule in Hoare's calculus. In our rule, the assumption predicate A_1 of the master specification serves as the invariant.

$$\frac{\begin{array}{l} A_1 \text{ is admissible} \\ A(i) \Rightarrow A_1(i, \langle\rangle) \\ A(i) \wedge A_1(i, x) \wedge C_1(i,x,o,y) \Rightarrow A_2(y) \\ A(i) \wedge A_1(i, z) \wedge C_1(i,z,o,y) \wedge A_2(y) \wedge C_2(y,z) \Rightarrow C(i,o) \\ A(i) \wedge A_1(i, x) \wedge C_1(i,x,o,y) \wedge A_2(y) \wedge C_2(y,z) \Rightarrow A_1(i, z) \end{array}}{[A,C] \trianglerighteq [A_1,C_1] \oplus [A_2,C_2]}$$

Figure 2 Master/slave refinement rule

The idea behind the rule is as follows. Assume the stream i fulfils the assumption predicate *A* of the original component. Initially the feedback stream x is empty, and the invariant A_1 is valid (second premise). Therefore the master produces output in the streams *o* and *y* that fulfils its commitment C_1. Because of the third premise, the assumption A_2 of the slave component is also fulfilled. The slave's output *z* fulfils the commitment predicate C_2, and because of the fifth premise the assumption A_1 stays valid, when the slave's output is fed back. This process goes on, until a stable situation, the fixpoint, is reached. Then the streams *x* and *z* are identical, and the fourth premise implies the commitment of our original component. The fixpoint, however, may be infinite. The purpose of the first premise is to ensure that the iteration process is valid even for infinite computations. A predicate over streams is *admissible*, if it holds for a stream whenever it holds for all finite prefixes of it.

11.3 Application of the Method

In our approach we model each component of the production cell as a stream-processing function. The streams between the components describe the flow of metal blanks and control signals.

As an example for the use of Focus and its refinement rule system, we consider the elevating rotary table (ERT) of the production cell. We make a further simplification: our table moves only vertically. It is quite easy, however, to add rotation to our specifications. From [5] we take the following informal description:

- The ERT receives a metal blank.
- The ERT moves to its upper position.

- The ERT informs the robot that a blank is available.
- The ERT receives an acknowledgement that the robot removed the blank.
- The ERT moves to its lower position and awaits the next blank.

First we need to consider the messages that the ERT consumes and emits. We need a message to express that a new metal blank has arrived at the table, that the table should go up or down, and that the robot has removed a blank from the table. We use three sets of messages:

- $B = \{ Blank \}$ to model the flow of metal blanks.
- $C = \{ Up, Dn \}$ to model the command signal flow from the table's controller to its motor.
- $A = \{ Ok \}$ to model acknowledgements from the robot back to the table.

Whenever the ERT's motor has executed a command, it will send an acknowledgement of this command back to the controller. For the acknowledgement of a message m we write $ack(m)$. We extend this notation to sets of messages by:

$$ack(M) = \{ ack(m) \mid m \in M \}$$

11.3.1 Descriptive specifications

A decomposition of the production cell (we skip the details here) yields the following specification of the table (as well as similar specifications for the other components):

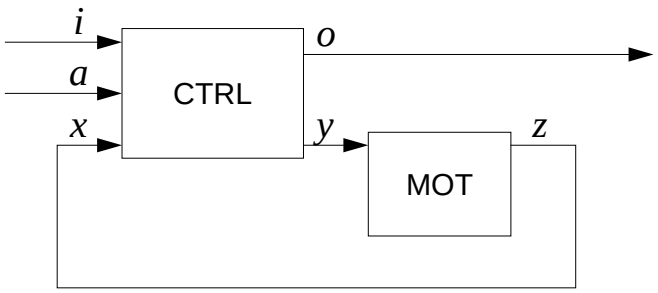
$$ERT = [\ \#a \leq \#i \leq \#a + 1,\ o = i\ ]$$

The streams i, a, and o are elements of B^ω, A^ω and B^ω, respectively. The table simply passes on every metal blank received on its input channel i (which is connected to the feed belt) to the output channel o (which is connected to the robot). The assumption of the specification states that the table is only required to pass on the blanks, if enough acknowledgements from the robot arrive via channel a (one for each blank, or possibly one for each except the most recent one).

The next refinement step decomposes the specification *ERT* into a control component specification *CTRL* and a component specification *MOT* that simulates the abstract behaviour of the ERT's motor:

$$ERT \succcurlyeq CRTL \oplus MOT$$

The result of the refinement is shown below. The streams *i*, *a*, and *o* are as above; *y* is an element of C^{ω}, and *x* and *z* are elements of $ack(C)^{\omega}$.



$$CTRL = [\, A_{CTRL}, C_{CTRL} \,]$$
$$A_{CTRL}(i,a,x) \equiv \#a \le \#i \le \#a + 1 \wedge$$
$$\#ack(Up)\copyright x \le \#i \wedge \#ack(Dn)\copyright x \le \#a \wedge$$
$$x \angle \langle ack(Up), ack(Dn) \rangle^{\infty}$$
$$C_{CTRL}(i,a,x,o,y) \equiv \#o = \#ack(Up)\copyright x \wedge$$
$$\#Up\copyright y = min(\#i, 1 + \#ack(Dn)\copyright x) \wedge$$
$$\#Dn\copyright y = min(\#a, \#ack(Up)\copyright x) \wedge$$
$$y \angle \langle Up, Dn \rangle^{\infty}$$
$$MOT = [\, \textbf{true},$$
$$z = map.\{\, Up \rightarrow ack(Up), Dn \rightarrow ack(Dn) \,\}.y \,]$$

The motor component *MOT* simply acknowledges the commands to go up and down. The commitment of the control component C_{CTRL} describes the operation of the controller:

- Whenever the motor component signals that the table reached its upper position, the blank is offered to the robot (via channel *o*). Since *o* only consists of messages *Blank*, it is sufficient to specify the length of the stream.
- A command is sent to raise the table for each blank that arrives, provided that the table is in its lower position (i.e., the motor returned a sufficient number of *ack(Dn)* messages).
- A command is sent to lower the table again whenever the robot signalled via channel *a* that it removed a blank, and the table is in its upper position.

- The controller alternately sends *Up* and *Dn* and starts with the message *Up*. This expresses an assumption for the cell: initially the table is in its lower position.

The controller's assumption predicate A_{CTRL} contains the assumption of the specification *ERT* (that there are enough acknowledgments from the robot) as well as some restrictions on the feedback from the motor:

- There are no more acknowledgements for the command *Up* than blanks arrive at the table.
- There are at most as many acknowledgements for the command *Dn* as blanks were removed by the robot.
- The acknowledgements arrive in the same order as the commands are sent by the controller.

This decomposition can easily be proven correct with our refinement rule for master/slave-systems.

11.3.2 Constructive specifications

So far we have an abstract specification: what we have basically described are the contents and length of the ERT's input and output streams. In the next refinement step we aim at a more operational characterisation of the table. In functional programming languages higher-order functions are often used to get shorter and clearer programs. Focus also allows higher-order functions, and indeed the function *map* we have introduced before allows us to concisely express the behaviour of the control component: incoming messages are converted to different messages and output again. Using two *map*-functions in parallel, we can handle the two output channels of *CTRL*, but what about the three input channels? We solve this problem with a separate SPF that merges the input streams into one. We call this function *zip*, as it zips its three input channels together. While in general this leads to some interesting problems (see [3] for a discussion), in our case it is easy because we know the order in which the messages arrive: first a blank, then the acknowledgement that the table has reached its upper position, then the signal that the robot has removed the blank and finally that the table is again in its lower position. We pass this information to the function *zip* in form of a sequence of numbers. This is the sequence of the channels to read the next message from: 0 is channel *i*, 1 is channel *a*, 2 is channel *x*. We skip the definition of this function and only show how it can be used for the constructive specification of the ERT's controller:

$$CRTL_{CONS} = [\, A_{CONS}, C_{CONS} \,]$$

$$A_{CONS} = A_{CTRL}$$

$$C_{CONS}(i,a,x,o,y) \equiv \exists h \in (B \cup A \cup ack(C))^{\omega} :$$
$$h = zip.\langle 0, 2, 1, 2\rangle.(i,a,x) \wedge$$
$$o = map.\{\, ack(Up) \rightarrow Blank \,\}.(ack(Up) \copyright h) \wedge$$
$$y = map.\{\, Blank \rightarrow Up, Ok \rightarrow Dn \}.(\{\, Blank, Ok \,\} \copyright h)$$

The assumption predicate of the controller is the same as above. Only the commitment has been changed. Again, the validity of the refinement

$$CTRL \succcurlyeq CRTL_{CONS}$$

can be shown using a refinement rule from [7] and predicate calculus.

11.3.3 Executable programs

We now have a constructive specification for the ERT. How can we transform this specification to an executable program? Up to now, Focus offers two target languages, an applicative language for abstract programs (AL), and a procedural language for concrete programs (PL). However, so far neither is implemented. We therefore chose to use Concurrent ML as the target language for the executable programs. Concurrent ML is an extension of ML with primitives for synchronous communication, but it is easy to implement asynchronous communication with unbounded buffers and the functions *map* and *zip*. We implement the function *map* using associative lists (the operator .-+->. constructs an association) so that elements not in the domain of the function are always removed from the input stream. Then the simulation program for the ERT closely resembles the commitment of the constructive specification:

```
fun ertmot i = mapper [“Up” -+-> ack “Up”, “Dn” -+-> ack “Dn”] i;
fun ertzip (i1,i2,i3) = zip [0,2,1,2] [i1,i2,i3];
fun ertctrl i =
      (mapper [ack “Up” -+-> “Blank”] ||
       mapper [“Blank” -+-> “Up”, “Ok” -+-> “Dn”]) i;
fun ert (i1,i2) = let
      val h1 = channel()
      val h2 = ertzip(i1,i2,h1)
      val (h3,h4) = ertctrl h2
      val h5 =ertmot h4
    in
      h5 ->- h1;
      h3
    end;
```

The infix function ->- connects two channels. Here we used strings as messages, because it simplifies the code to generate traces of the system. Of course, Concurrent ML supports the full set of ML data types for communication.

Since so far Concurrent ML has no denotational semantics, we cannot prove the correctness of our implementation, but we believe that the informal semantics of the language are close enough to our formalism to justify this step. True control programs for the ERT can be developed by further refinement of the specification *MOT* that yields a second master/slave-system with a low-level control component as master and the table's actuators and sensors as slave component.

Functional programming languages are not yet of industrial interest, but simple programs such as this one can be transformed into procedural languages using transformations rules, as explained in [1].

11.4 Conclusion

The remaining components can be refined in a similar way, yielding specifications and programs for the complete system. In our modeling we made some assumptions about the behavior of the cell to simplify the specifications. Most of them only concern initial positions of the machines; two more restrictive assumptions are the following:

- The press is initially non-empty. This allows an easier description of the robot‘s control cycle.
- The feed belt knows the time it takes for a blank to reach the ERT, and only then sends the message *Blank* to the ERT. We could specify this behavior by extending Focus to deal with real-time aspects. This area, however, is still part of ongoing investigations.

How well does our system fulfill the requirements of the informal task description? We guarantee that the production cell is alive, by demanding that the complete cell behaves basically like the identity function: everything that goes in, must come out again. This property is maintained throughout all refinement steps.

The task description contains two kinds of safety requirements. Local requirements are concerned with the sequence of control commands sent by the controllers, and of the acknowledgements sent back by the controlled hardware. We specify and verify these requirements whenever we introduce a new component. Further refinement steps leave them intact.

Because of the assumption/commitment-style of our specifications, in many cases it is possible to simplify the specifications of slave components, because they can rely on their master to send commands in the proper sequence. The second kind of safety requirements are global requirements about components that are not directly connected. For example, a robot arm may not pass the closed press, when it is extracted. For global safety requirements (and also for more general liveness properties than the simple deadlock-freedom in our work) it is necessary to begin development at the level of trace specifications, where only safe system runs are specified. The SPECTRUM specification in [11] is an example of such a trace specification. To verify that a design specification fulfills a trace specification, and therefore also the specified safety properties, we have to show that the set of traces generated by stream processing functions is a subset of the traces specified by the trace specification. While in general it is somewhat difficult to prove this inclusion, in many cases it is possible to derive a first design specification from a trace specification by syntactic transformations, as shown in [9].

We believe that the main advantage of Focus is its wide range of applications. It supports the complete development process starting with a very abstract specification and ending up with an implementation. A further advantage is the modularity of Focus. The specification can easily be adapted for other production cells using the same or other similar machines. In the decomposition of the specification for a new cell only the material flow has to be considered. Since the individual components of a distributed system are only loosely coupled, the internal operations of each machine are irrelevant for the decomposition.

The final specification of the production cell has a length of about 80 lines. We are sure that even if the restrictions mentioned above were removed, the specification would remain compact. This is due to the use of the assumption/commitment-style as well as higher-order functions in specification.

References

[1] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The design of distributed systems - an introduction to Focus.Technical report SFB 342/3/92 A, Technical University Munich, 1992

[2] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. Summary of case studies in Focus - a design method for distributed systems.Technical report SFB 342/3/92 A, Technical University Munich, 1992

[3] M. Broy. Functional specification of time sensitive communication systems. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness. Lecture Notes in Computer Science 430*, pages 153-179. Springer, 1990.

[4] G. Kahn, The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Information processing 74*, pages 471-475. North-Holland, 1974

[5] T. Lindner, A. Rüping, and E. Sekerinski. Aufgabenstellung für die Fallstudie “Fertigungszelle”. Internes Arbeitspapier, Forschungszentrum Informatik, Karlsruhe, 1992.

[6] J. Philipps. Spezifikation einer Fertigungszelle - Eine Fallstudie in Focus. Diploma Thesis, Technical University Munich, 1993.

[7] K. Stølen, F. Dederichs, and R. Weber. Assumption/commitment rules for networks of asynchronously communicating agents. Technical report SFB 342/2/93 A, Technical University Munich, 1993

[8] M. Broy. Towards a Formal Foundation of the Specification and Description Language SDL. Formal Aspects of Computing, Vol. 3, p. 21-57, 1991.

[9] C. Dendorfer and R. Weber. From service specification to protocol entity implementation - an exercise in FOCUS. Technical Report SFB 342/4/92 A, Technical University Munich, 1992.

[10] M. Fuchs. Technologieabhängigkeit von Spezifikationen digitaler Hardware. Technical Report SFB 342/14/94 A, Technical University Munich, 1994, PhD-Thesis.

[11] C. Lewerentz, T. Lindner. Case Study Production Cell. LNCS, Springer, 1994.